NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

REFINING A TASK-EXECUTION TIME PREDICTION MODEL FOR USE IN MSHN

by

Blanca A. Shaeffer

March 2000

Thesis Advisor:	James Bret Michael
Second Reader:	Mantak Shing

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 2000	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE: Refining A Task-Execution Time Prediction Model For Use In MSHN	5. FUNDING NUMBERS
6. AUTHOR(S) Shaeffer, Blanca A.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT: Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE:

13. ABSTRACT *(maximum 200 words)*

Nowadays, it is common to see the use of a network of machines to distribute the workload and to share information between machines. In these distributed systems, the scheduling of resources to applications may be accomplished by a Resource Management System (RMS).

In order to come up with a good schedule for a set of applications to be distributed among a set of machines, the scheduler within an RMS uses a model to predict the execution time of the applications. A model from a previous thesis was analyzed and refined to estimate the time that the last task will be completed when scheduling several tasks among several machines. The goal of this thesis was to refine the model in such a way that it correctly predicted the execution times of the schedules while doing so in an efficient manner.

The validation of the model demonstrated that it could accurately predict the relative execution time of a communication-intensive, asynchronous application, and of certain compute-intensive, asynchronous applications. However, the level of detail required for this model to predict these execution times is too high, and therefore, inefficient.

14. SUBJECT Resource Management System, Operating Systems, Distributed Systems, Scheduling, MSHN, Heterogeneous Computing	15. NUMBER OF PAGES 94
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

Approved for public release; distribution is unlimited.

REFINING A TASK-EXECUTION TIME PREDICTION MODEL FOR USE IN MSHN

Blanca A. Shaeffer
Lieutenant, United States Navy
B.S., United States Naval Academy, 1995

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2000

Author: Shaeffer
Blanca A. Shaeffer

Approved by: James Bret Michael
James Bret Michael, Thesis Advisor

Mantak Shing , Second Reader

Dan C Boger
Dan Boger, Chairman, Department of Computer Science

ABSTRACT

Nowadays, it is common to see the use of a network of machines to distribute the workload and to share information between machines. In these distributed systems, the scheduling of resources to applications may be accomplished by a Resource Management System (RMS).

In order to come up with a good schedule for a set of applications to be distributed among a set of machines, the scheduler within an RMS uses a model to predict the execution time of the applications. A model from a previous thesis was analyzed and refined to estimate the time that the last task will be completed when scheduling several tasks among several machines. The goal of this thesis was to refine the model in such a way that it correctly predicted the execution times of the schedules while doing so in an efficient manner.

The validation of the model demonstrated that it could accurately predict the relative execution time of a communication-intensive, asynchronous application, and of certain compute-intensive, asynchronous applications. However, the level of detail required for this model to predict these execution times is too high, and therefore, inefficient.

TABLE OF CONTENTS

I. INTRODUCTION

This thesis explores the use of an analytical model to predict an application's performance given a particular distributed network system and an accurate assessment of the available resources. The goal of this thesis is to refine an already-existing model to more accurately predict the execution times of applications. A previous thesis investigated questions similar to those addressed here, but limited its scope as follows:

- The use of three identical machines to form the network;
- Each machine consisted of Pentium processors, each running either Linux Kernel 2.0.32 or Microsoft Windows NT Workstation 4.0 as the operating system (the same operating system on all three machines at the same time);
- The three machines were connected by an isolated local area network;
- Each application consisted of three inter-communicating processes, each of which executed within a Java Virtual Machine (JVM);
- Each of the three processes consisted of five threads;

- The only processes running on the machines at the time of data-collection were those that applied to the research;
- All processes ran either in a synchronous or an asynchronous mode;
- All processes were compute-intensive.

The model, as it currently stands, correctly predicts the relative execution times of compute-intensive applications, but it does not accurately predict the performance of communication-intensive applications. The model's predictions do not account for the following:

- The time required for the CPU's to switch from one thread of execution to another (context switching). The more threads that require the use of each CPU, the more time it takes to perform a context switch;
- The time that is required for a process to "recover" from a collision on the network (when two processes attempt to send a message over the network at the same time);
- The overlap of a process's communication time and its computation time.

The model uses an analytical solution that predicts how long a given application will run when given certain resources to use. In order to test the correctness of the model, an application emulator was developed to emulate computationally intensive and communication-intensive, as well as synchronous and asynchronous, applications.

The next section of this chapter will give some background that explains the motivation for conducting this thesis research and how the research furthers the goals of a project called MSHN (Management System for Heterogeneous Networks). Section B will explain the scope of the model and the emulator and why they were chosen. The last section will outline the remainder of the thesis.

A. BACKGROUND

1. MSHN

A **computer network** is a collection of autonomous computers that are able to communicate with each other. This is much different from a **distributed system**, where multiple autonomous computers are available to a user, but the way in which the computers are utilized is transparent to the user. This means that there is some sort of software in between the user and the distributed system that chooses what resources to assign to what jobs.

When assigning resources to an application on one machine, that machine's operating system handles every aspect of the scheduling to ensure that applications run as efficiently as possible. By "resource," we mean anything, including CPU time, that an application may need to successfully complete its assigned task. Other examples of resources are input/output devices and memory.

Nowadays, it is common to see the use of a network of machines to distribute the workload and to share information between machines. In these distributed systems, the task of assigning resources to applications becomes a problem of larger scope than that of the operating system on a single machine, although the concept is quite similar.

A Resource Management System (RMS) is similar to a distributed operating system in that it will distribute available resources to the applications that need them, and will sometimes even break up an application so that the separate parts can be assigned separate resources and increase the performance of the whole application. However, a RMS differs from a distributed operating system in that it does not micro-manage each computer's resources. Each computer runs its own operating system, as all other resources run their own protocols or operating system. The RMS is responsible for keeping track of the status of all resources and applications and for issuing commands to begin executing applications.

The purpose of the MSHN project, from which this thesis was borne, is to develop a RMS that supports the execution of many different kinds of applications, each with its own requirements, in a distributed and heterogeneous environment [Ref. 1]. The functions to be performed by MSHN are the following:

- monitoring general resource availability,
- transparent sensing of resource requirements of applications,
- on-line measurement of resource and system state,
- mapping tasks and subtasks onto a heterogeneous suite of machines in a way that exploits heterogeneity,
- adaptation of jobs to variations in the availability of resources. Factors influencing QoS for MSHN include security, deadlines, priority, adaptability (preferences for different versions), resource availability, and external users,
- meeting QoS requirements including real-time deadlines, fault tolerance, security, and priorities.

This thesis is an important part of the MSHN project, which is part of Defense Advanced Research Projects Agency's

(DARPA) Quorum program. The QUORUM program has as its overall goal the determination of whether next-generation Aegis and C4I applications, because of their diverse needs, which include some real-time needs, can be supported by Commercial Off-the-Shelf (COTS) and Government Off-the-Shelf (GOTS) products. MSHN's role is to define a resource management system that would wisely allocate the COTS/GOTS resources to such diverse applications.

2. Scheduling

When scheduling resources for use by applications, each resource can only be assigned to one application at a time. For example, two processes running on a multitasking system must share the CPU, with each running for a specific amount of time before the other interrupts it (i.e., time-slicing).

There are different methods of scheduling, each of which is more appropriate for certain situations than the others. One such method is that of a *queue*. When using a queue, each program waits its turn and is executed in a serial fashion, usually on a first-come-first-served (FIFO) basis. The queue is the most appropriate method for scheduling devices such as printers, where each different print job must be completed before the next can be started.

Another method used in scheduling is called a *round-robin*. The scheduler will give each job a short time-slice of processing time before moving on to the next job. When using this approach, all jobs advance in small steps until the last job is completed.

In a distributed system, applications are scheduled in a way that will maximize the use of the resources available. In order to be even more efficient, an application may be divided up such that each part can be executed on different machines.

When a RMS schedules its applications, there are multiple considerations it must make. MSHN uses a Scheduling Advisor to determine which set of resources to assign to a newly arrived process. The Scheduling Advisor needs to know which resources and how much of those resources a process will need in order to meet its Quality of Service (QoS) requirements. To aid the Scheduling Advisor, MSHN uses a Resource Requirements Database that stores very fine grain information about the resource usage of each application that was previously executed by the RMS [Ref. 1].

In order to test the RMS and all of its components, it is necessary to run many different types of applications on the system. It would have been very expensive and timeconsuming to build or acquire the many different applications needed. Therefore it was decided that a general-purpose application emulator, whose parameters could be changed to cause it to emulate the different applications, was the best recourse.

B. APPLICATION EMULATOR AND PREDICTION MODEL

1. Emulator Functionality

A real application running on a distributed system will require the use of certain resources over others. In general, an application can either be compute- or communication-intensive. When an application is compute-intensive it will need a lot of CPU processing time in order to accomplish its task. On the other hand, if an application is more communication-intensive, it will need to be allocated communication devices in order to accomplish its task.

The application emulator will emulate compute- and/or communication-intensive applications and leave the same "resource usage footprint"(Ref.1) as the real application would leave. The emulator allows for the simulation of running applications without having to acquire, install, and maintain the real applications.

The second purpose for the application emulator is that it will act as a monitor when there are no other MSHN-scheduled applications running on the system. While no applications are being executed on the system, it is still necessary to identify the status of the resources available. In order to do this, an instance of the application emulator will be started. This will be a default occurrence at start-up of the system.

2. Prediction Model

In order to come up with a good schedule for a set of applications to be distributed among a set of machines, it would be helpful to be able to predict how long an application will run on a given machine. The task of predicting the run-time of a certain application is not a trivial problem and is known as the "execution time estimation problem" (Ref. 3).

There are three methods that can be followed when solving the execution time estimation problem: code analysis, analytic benchmarking/code profiling, and statistical prediction. Code analysis requires the thorough study of the source code of the application. For all but trivial problems, this is an inefficient method, and is not conducive to a heterogeneous computing environment.

Analytic benchmarking/code profiling is more useful in a heterogeneous computing type of environment because it identifies different primitive code types and then, for each code type, it obtains performance benchmarks for each machine in the heterogeneous system. Code profiling will determine the code types of which a certain application is made up. The combination of the code profiling and the benchmarking together produce an estimate of the application's execution time.

The last of the methods, statistical prediction, makes use of the applications' execution times in previous runs. As the number of times the application is executed increases, odds are that the accuracy of the prediction will also increase. The problem with this method is that there is no way of predicting the execution time of a given application if it has never been executed on a certain machine.

The model used for this thesis, which estimates the time that the last task will be completed when scheduling several tasks among several machines, uses an analytical, closed-form solution to solve the problem. This is a form of the benchmarking/code profiling technique mentioned earlier. The model will be explained in more detail in Chapters II and III.

C. ORGANIZATION

Previous work done in this area of research will be discussed in Chapter II of this thesis. Chapter II covers in detail the thesis to which this thesis has followed up. It also presents some work performed in statistical prediction for distributed computing environments and how SmartNet, MSHN's predecessor, handled its scheduling. Chapter III will discuss the details of the prediction model, as well as how the application emulator is used to validate the model. Chapter IV will present the results of our experiments. Finally, Chapter V will give a summary of the thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

II. RELATED WORK

This chapter consists of a presentation of the results of the thesis to which this thesis follows up. As described in Chapter I, the motivation for this research is to determine the granularity necessary for a model to predict execution times of applications to be scheduled in a heterogeneous distributed computing environment. The prediction task raises some fundamental issues. However, very little sophisticated research has been conducted to address these issues. We will discuss some recent research conducted at Ohio State University that addresses this very topic. The last section of this chapter summarizes the functionalities of another resource management system, SmartNet, which is the predecessor of MSHN.

A. FIRST VERSION OF MODEL

1. Goal of Model

The goal of the model as it was originally designed was to "determine the appropriate granularity to use in a model for resource allocation."(Ref. 1) This means finding a good balance between finding a model that is so fine-grained that it takes a long time to produce its result, and finding a model that is too simple as to produce a result that is either not accurate enough or completely incorrect. There are trade-offs between using a fine-grained model and using a coarser grained model. The overhead incurred and the desired level of accuracy are just two of the many aspects of choosing the right level of granularity in a model.

As was listed in Chapter I, this execution time prediction model was able to predict the relative execution time of an application that was compute-intensive, but not that of a communication-intensive application. There are times when determining the relative performance of a schedule is "good enough," but there are times when it is necessary to predict the absolute performance. The following example of the difference between relative and absolute performance was given in Reference 1.

Say we have two schedules of an application, each with a predicted and an actual run time, as shown in Figure 1. If we have accurately predicted the relative performance of two schedules, when the predicted times, A1 and B1, are compared to the actual run times, A and B, we can correctly determine which schedule is better than the other. In this example, if we choose schedule A, we can deem the prediction model "good enough" for our purposes. However, if we choose schedule A when assigning a particular task to a machine, and the same request is made again, our model would again assign the same schedule for that second task. This is because the time required to execute two A1 schedules is less than the time required for one B1 schedule to execute. But in reality, twice the time needed for schedule A is not less than the time required for schedule B.

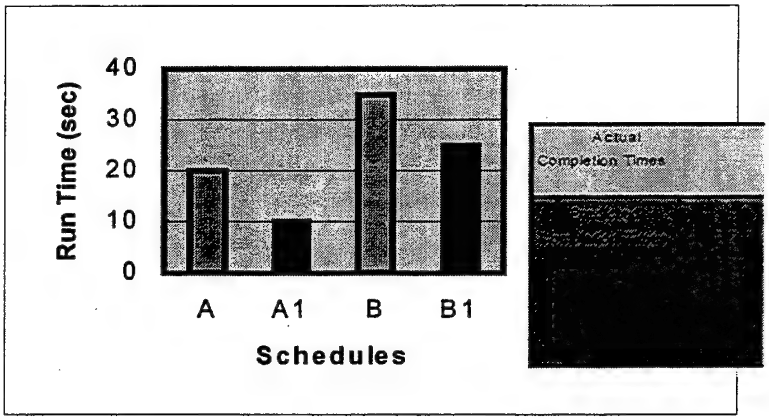


Figure 1 [From Ref. 1]. Example of Actual v. Predicted Times

2. Model Composition

This model uses as its heuristic the time at which the last process in a schedule completes its execution. The class of applications that can be modeled using this simple model is that of applications whose resource usage patterns can be divided up between its computation and its communication load. This allows the model to remain flexible in the way that it defines a particular application, while still being able to model a wide variety of applications. (Ref. 1)

The inputs to the model, known *a priori*, are broken down as follows:

a) Computation Time

- The amount of time the application is expected to use the CPU.
- This does not include any time the CPU is used for process communications.
- If more than one process is using a CPU at the same time, the computation time must be dilated.

b) Communication Time

- This is divided up into the time spent sending the message, which is dependent on the throughput of the communication link (either an Ethernet connection or shared memory within a single machine) and the time spent processing the message by the sender and receiver (considered the latency time). The algorithms for measuring the throughput and latency between/within machines are described later.

- If more than one process is sending data over any communication link, then there is a contention for that resource.
- We know the mean number of messages, along with the mean size of those messages and their distribution types, that each process is to send to each of the other processes.

3. Measuring Latency

The time required for a message to propagate up and down the TCP/IP stack is an example of the network latency as used in this model. To measure this value, we use the following method: with no other network traffic currently on the links, we send a large number of small messages (in this instance, 10,000 1-byte messages), and then echo the messages back to the sender. This time represents two latency times, since we measure the time it took for the receiver to receive and then send the messages back to the sender. If this latency time is divided by two, then we have our one-way latency. Now this value, divided by the total number of messages, gives us the latency per message.

This latency, when actually measured and compared to the propagation time, was too insignificant to include in the model. It was therefore not actually used.

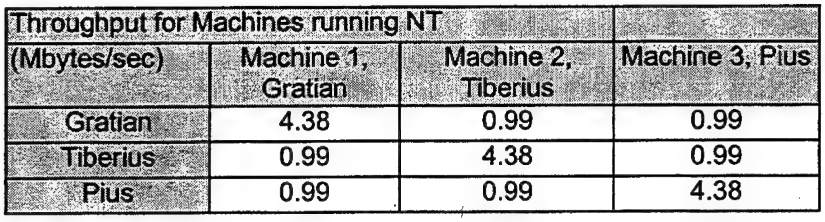
4. Measuring Throughput

The model uses a procedure that sends a large message between the sender and the receiver (both could be on the same or on different machines). This is different than the method used for measuring the latency, which sent a large number of very small messages, assuming that the propagation time for very small messages is zero. The time required for the message to be sent, received, and echoed back to the sender is measured. That time divided by two gives you the time required to send the message in only one direction. We then subtract the one-way latency time, since it is only the propagation time for which we are looking. If that time is divided by the size of the message sent, the resulting value is the throughput for that link.

5. Model Validation

In order to validate the model, the application emulator mentioned in Chapter I was used. This emulator takes in the "model parameters as input and emulates the behavior of an application." (Ref. 1) These applications consist of processes that communicate with each other and also spend some time performing computations. By using the input parameters that the prediction model uses as inputs to the emulator and measuring how long the applications actually take to run, it is possible to then compare the predicted versus the actual run times.

Table 1 shows the throughput measured, and therefore the values used for the predictions in the model.



Throughput for Machines running NT			
(Mbytes/sec)	Machine 1, Gratian	Machine 2, Tiberius	Machine 3, Pius
Gratian	4.38	0.99	0.99
Tiberius	0.99	4.38	0.99
Pius	0.99	0.99	4.38

Table 1 [From Ref. 1]. Measured Throughput

Table 2 represents the input parameters to the emulator in order to measure the CPU time of one process, with sole use of the CPU, without sending any messages. Table 3 shows the average time for an application's computation time. The times are for all processes running on either a Windows NT or Linux operating system, and either in a Graphical User Interface (GUI) version or in a non-GUI version.

Number of passes through matrix multiply:	1
Number of messages sent (on average):	0
Distribution of time between messages:	Constant
Size of message (on average):	4k bytes
Distribution of message size:	Constant
Synchronous	1

Table 2 [From Ref. 1]. Parameters Used For CPU Measurement

Average CPU only time (seconds)		
	Linux	NT
GUI version	14.08	2.65
non-GUI version	14.01	1.51

Table 3 [From Ref. 1]. Average CPU Only Time

Four experiments were conducted to compare the model to the actual emulator execution times. In each of the experiments, the emulation of the application consisted of three homogeneous processes, each with a computational thread and an input and output thread for each of the two other processes. All of the output threads sent the same number of messages, all of the same distribution in size, to the other processes. The three processes, distributed among the three machines, make up 27 possible schedules. The mapping of processes to the machines is as shown in Table 4.

Schedule Number	Machine Assignment
1	111
2	211
3	311
4	121
5	221
6	321
7	131
8	231
9	331
10	112
11	212
12	312
13	122
14	222
15	322
16	132
17	232
18	332
19	113
20	213
21	313
22	123
23	223
24	323
25	133
26	233
27	333

Table 4 [From Ref. 1]. Mapping of Schedule Number To Machine Assignments

The experiment details were as follows:

a) *25 Messages, Synchronous, GUI Version*

Table 5 shows the parameters used as inputs for the model and emulator. As shown in Figure 2, the model was able to predict the relative run times of all possible schedules of the three processes on the three machines. However, the model is not able to predict absolute run times.

Number of passes through matrix multiply:	1
Number of messages sent (on average):	25
Distribution of time between messages:	constant
Size of message (on average):	4k bytes
Distribution of message size:	constant
Synchronous	1

Table 5 [From Ref. 1]. Parameters Used for Experiment One

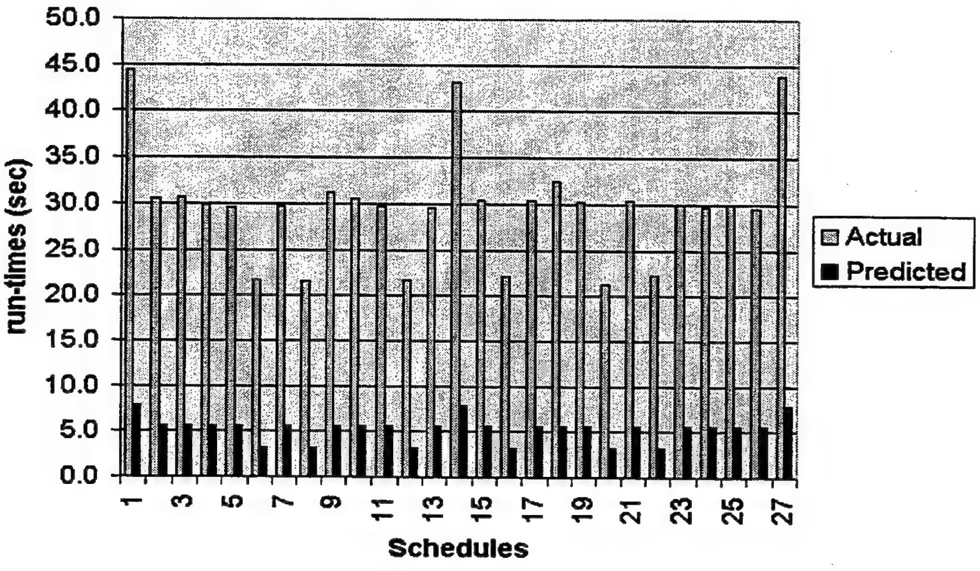


Figure 2 [From Ref. 1]. Actual vs. Predicted Run-Times for NT, Experiment One, GUI, 25 Messages, Synchronous

b) 25 Messages, Synchronous, Non-GUI

The parameters used as inputs to the emulator for this experiment are the same as those used for experiment one. The only change between the two experiments was the use of the GUI- versus the non-GUI version of the emulator. The model was still not able to predict absolute run times, but relative performance was still predicted correctly. Figure 3 shows the results of experiment two. Note that the scale of the y-axis is different for each experiment's output graph. This is due to the different range of run times for the schedules in each experiment.

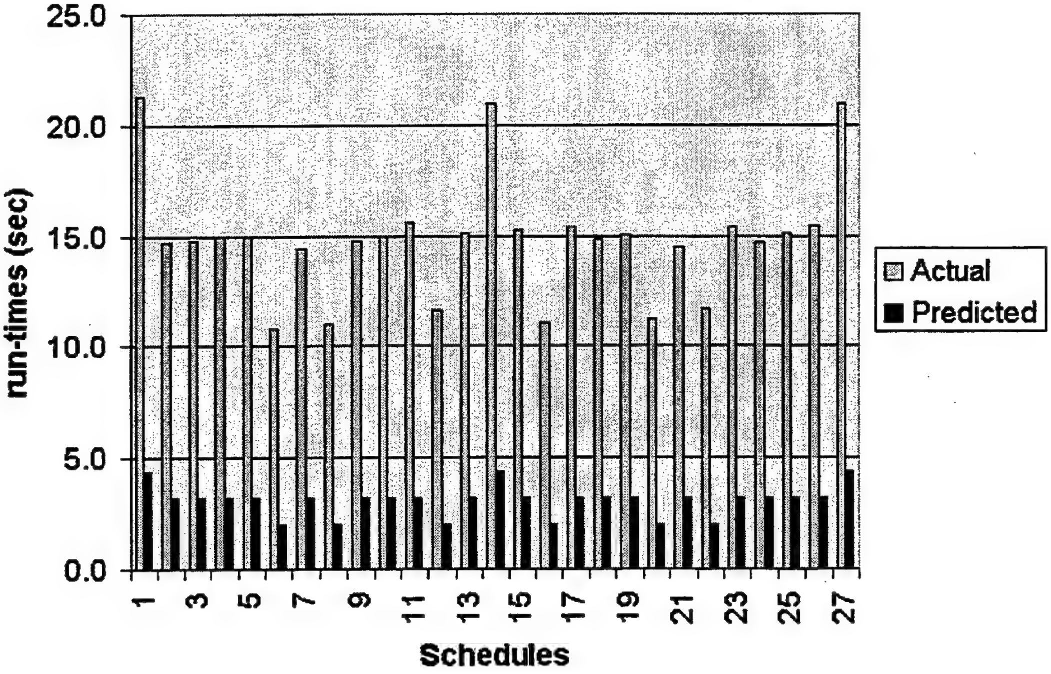


Figure 3 [From Ref. 1]. Actual vs. Predicted Run-Times for NT, Experiment Two, non-GUI, 25 Messages, Synchronous

c) 25 Messages, Asynchronous, Non-GUI

This third experiment uses the same input parameters as experiment two, except that the messages sent from each process to the other processes are sent asynchronously. Figure 4 shows the improvement in the performance of the application when all messages are not sent synchronously, and the performance of the model is also much improved. The gap between the predicted and the actual run times was lessened noticeably.

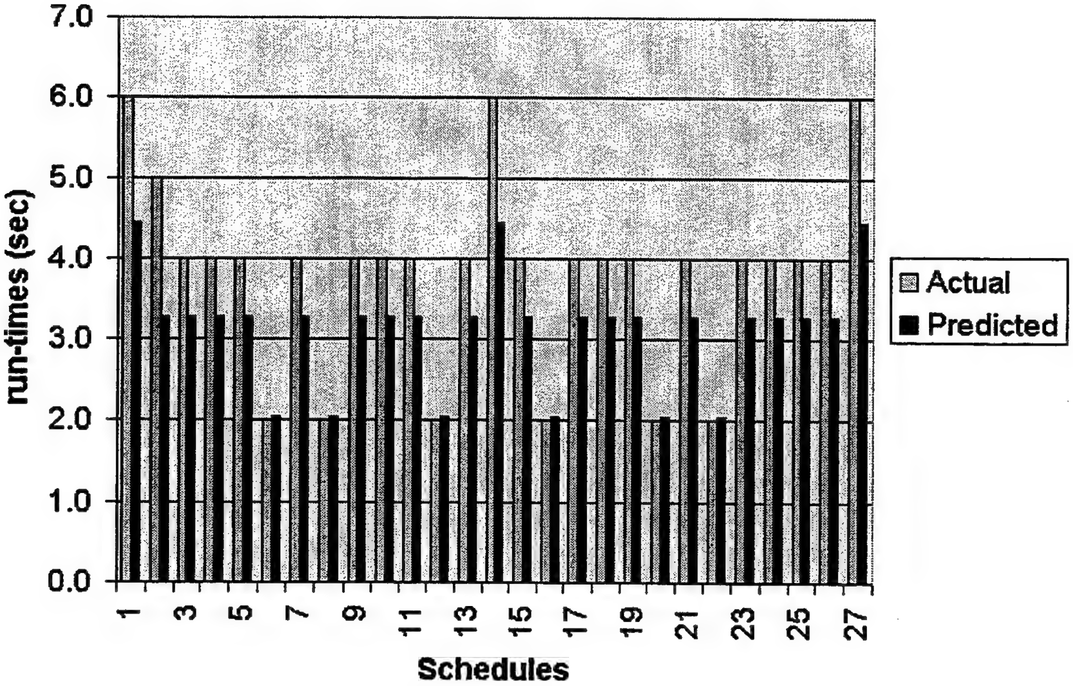


Figure 4 [From Ref. 1]. Actual vs. Predicted Run-Times for NT, Experiment Three, non-GUI, 25 Messages, Asynchronous

d) 1250 Messages, Asynchronous, Non-GUI

After running three experiments that emulated more compute-intensive applications, this last experiment was aimed at emulating a more communication-intensive application. The results, shown in Figure 5, tell us that the prediction model breaks down when attempting to predict run-times for communication-intensive applications. Not only does it not correctly predict absolute performance, but it also does not correctly predict relative performance.

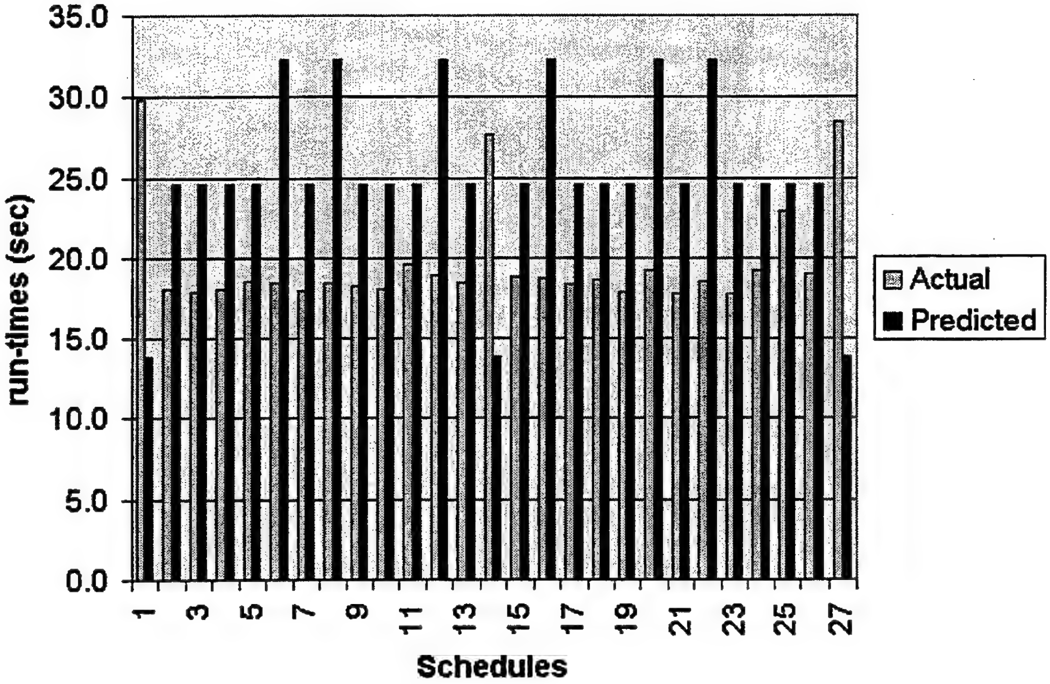


Figure 5 [From Ref. 1]. Actual vs. Predicted Run-Times for NT, Experiment Four, 1250 Messages, Asynchronous

B. STATISTICAL PREDICTION THROUGH ANALYTIC BENCHMARKING

This method of predicting task execution times for use in a distributed heterogeneous environment was developed at Ohio State University. The advantages of adopting this method are that it does not need to know *a priori* how long an application takes to run on each available machine in order for a good schedule to be determined, and that the algorithms can take into account the differences between each machine's capabilities. This method defines a task's execution time as a random variable, and computes that random variable as a function of the input data and machine capabilities for that specific task on a specific machine [Ref. 3].

Any process has an input data set that determines its execution time on a certain machine. This input data set can be described as a parameter vector such as:

$$X = \left[x^1 x^2 \cdots x^p \right].$$

The function $t = m(X)$ can be used to model the task execution time according to this parameter vector. It is not always the case, however, that all q parameters can be modeled, thus only allowing p parameters, where $p \le q$, to be modeled. Now the function used to model the execution time of a task according to its input parameter vector becomes the random variable:

$$t = m(X) + \varepsilon,$$

where $m(X)$ represents all the modeled factors affecting the execution time and ε represents all the unmodeled factors.

The study done at Ohio State University was divided into two distinct parts: the first computes the execution times of a task given an input parameter set on a particular machine, while the second also uses a parameterization of different machines. By adding the information of the machine type, the task execution time can be modeled as a function of not only previous observations of the task executing on machines of the same type, but on all observations of previous executions, regardless of the machine type.

1. Observations Between Different Machine Types are not Shared

In the first part of this study, when given an input parameter vector, *X*, and a given task, the authors present a method for estimating $\hat{m}(X)$ *and* $\hat{\varepsilon}$, estimators for $m(X)$ and ε in the equation presented earlier. For the given task, there will exist a set of n previous observations of the execution time $\{(t_i, X_i)\}_{i=1}^{n}$, where t_i is an observed execution time for the parameter vector X_i [Ref. 3]. Each machine type in the heterogeneous network requires a separate set of observations to be maintained and used in the statistical prediction of the current task's execution time. Thus, if a task is executed on a given machine type n times, and then the same task is scheduled to run on a different type of machine, those n observations cannot be used in the prediction of the new execution time.

Nonparametric regression techniques are used to compute $\hat{m}(X)$, since the estimate depends only on the set of previous observations. This technique uses the equation

$$\hat{m}(X) = \frac{1}{n}\sum_{i=1}^{n} W_i(X)t_i.$$

W_i is a weighting function that assigns higher weights to observations close to the parameter X than to those farther from X, as shown in Figure 6. This equation states that for a certain parameter vector X, $\hat{m}(X)$ is a weighted average of the execution times, t_i, of the past n observations.

2. Parameterizing Machine Performance

The second section of this study incorporates all of the sharing of observations between different machine types. The authors point out two reasons why one would want to have this capability:

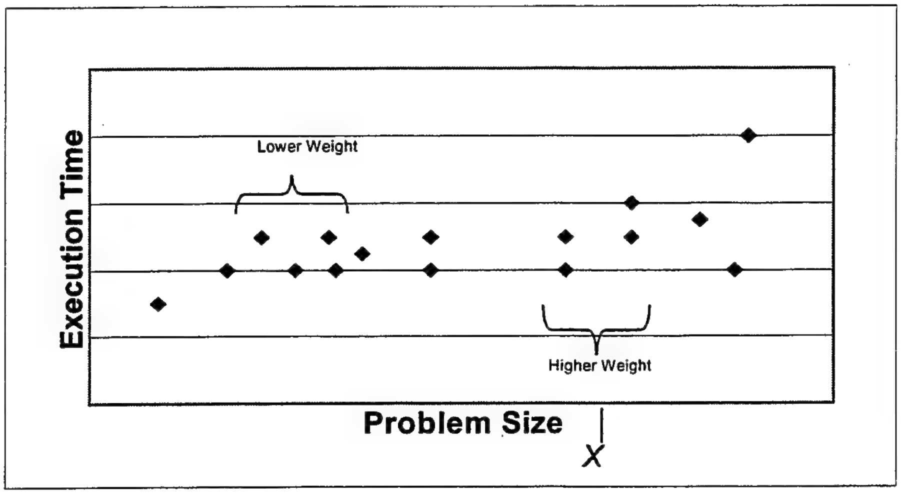


Figure 6. Assigning Weights to Observations

a) Requiring a different set of observations for each machine type makes it difficult to add or remove any new machines or applications to or from the network. The system will have to gather a few observations for each new machine-application pair for the algorithm to accurately predict the new execution times.

b) Because of the increased likelihood of the algorithm to coming up with incorrect prediction times when there are only a few observations for a given machine, there is a chance that the new machine will not get tasks scheduled for execution on that machine. If that new machine is starved of applications to run, then it will not gather new observations to add to its short list of observations, therefore not improving its chances of being chosen by the scheduler [Ref. 3].

In order to share all observations from all machines when predicting the execution time for an application on any given machine, one has to have a method for numerically characterizing all machines and thus include that characterization in the input parameter vector. This method would numerically indicate the performance difference/similarity between any given set of machines. The method of analytic benchmarking is used for this purpose.

Using this method, each machine in the network is characterized by a benchmark vector. Ideally, one would like to have a benchmark for every possible code type that characterizes the performance of each machine. In reality, it is not efficient, if even possible, to come up with a complete set of benchmarks. Therefore, a "reasonable set of r benchmarks [Ref. 3]" is obtained. This set of benchmarks will comprise a *machine space*, where each machine is represented within the machine space by a *benchmark vector*.

For example, machine i is represented by the vector $B_i = \left[b_i^1 b_i^2 \cdots b_i^r\right]$, where benchmarks *1* through r affect the performance of machine i.

To predict the execution time of a task with an input parameter vector of $X = \left[x^1 x^2 \cdots x^p\right]$ on machine i with a benchmark parameter vector $B_i = \left[b_i^1 b_i^2 \cdots b_i^r\right]$, the following parameter vector would be used: $Y = \left[b_i^1 b_i^2 \cdots b_i^r x^1 x^2 \cdots x^p\right]$.

3. Algorithm Summary

The following is the authors' summary of the Execution Time Estimation Algorithm presented in this paper (Ref. 3):

```
begin
    For each candidate machine $j$ with
    benchmark vector $B_j = \left[b_j^1 b_j^2 \cdots b_j^s\right]$

    begin
        Compute $\hat{m}(Y_j)$ and $\hat{\varepsilon}$ where
        $Y_j = \left[b_j^1 b_j^2 \cdots b_j^s x^1 x^2 \cdots x^p\right]$.
    end
    Give estimates computed above to
    matching and scheduling algorithm.
    The algorithm will return a machine $j$ chosen to execute the task.
    Execute the task on machine $j$ and measure the execution time $t_{n+1}$.
    Add observation $\left(t_{n+1}, Y_{n+1}\right)$ to the set of observations.
    $n = n + 1$.
end
```

When a new application is first introduced to the network, it must be run at least one time in order to get at least one observation. To function correctly, the algorithm requires each application to have at least one recorded observation. For the algorithm to function more precisely, the new application should be executed on at least a few of the machines. The authors state that "these values are easily obtained during the development, testing, and debugging of the application." (Ref. 3)

4. Validation of Algorithm

In order to validate the use of their algorithm, the authors ran a number of experiments using a network of 16 heterogeneous machines. In three different experiments, they simulated adding a new machine to the existing network and running an application on that machine. These simulations were done to compare the performance difference of the execution time estimation algorithm when observations can and cannot be shared between machines. The number of observations varied between 1 and 50. The application that was chosen to run was a Cholesky decomposition algorithm whose execution time depends on one parameter: the size of a matrix.

The first simulation shows the performance of the estimation algorithm when observations cannot be shared between machines. Therefore, the input parameter vector to the estimation algorithm was made up of only the size of the matrix used in the Cholesky algorithm task. The second simulation added the ability of the estimation algorithm to use 350 observations previously gathered from executing that application uniformly on the other 15 machines. In this simulation, a 10-dimensional machine space (constructed from 10 benchmarks), reduced to 3 dimensions, was used. The third simulation was similar to the second, with the difference being that the full 10-dimensional machine space was used.

The results of the three simulations show that being able to share observations between machines produces a much more accurate estimated execution time than when observations are not shared. However, the difference between the second and third experiments shows that there is not much of a difference between the performance of the algorithm when using a full or a reduced machine space. When the number of observations was low, the fist simulation produced prediction errors of, on average, around 500 percent. With the same number of observations, the second and third simulations produced errors of around 50 percent. When the number of observations grew larger, all three simulations produced errors of around 15 percent. The advantage of using a reduced machine space over a full machine space comes into play when measuring the computational cost of the estimation algorithm. Using a reduced machine space proves to be much more efficient. (Ref. 3)

C. SMARTNET

MSHN's predecessor, SmartNet, is a scheduling framework that has been successfully used by the Department of Defense and the National Institutes of Health. It was developed by the Heterogeneous Computing team at the US Navy's facility at the Naval Command, Control, and Ocean Surveillance Center (NCCOSC) for Research, Development, Testing and Evaluation (RDT&E) in San Diego [Ref. 2].

SmartNet's purpose was to optimize schedules for compute-intensive jobs among a network of heterogeneous computers [Ref. 1]. By using SmartNet as the scheduler or basic RMS within a distributed system, that computing environment's performance in executing its applications could be improved.

The major research contributions made by SmartNet included using a job's compute characteristics and data collected from the previous executions of the job to predict that job's expected run-time on a particular machine. With the ability to predict the execution times of applications, SmartNet was geared towards minimizing the time when the last job scheduled among a set of jobs finished running. SmartNet supported the idea that being able to estimate average application run-times was good enough for scheduling purposes, and that predicting exact run-times was not absolutely necessary [Ref. 1].

SmartNet consists of four separate processes:

1. Controller -- interfaces with and manages all resources.
2. Scheduler -- includes Exhaustive, Greedy, Evolutionary, and Simulated Annealing scheduling

algorithms. New algorithms are easily integrated with the existing ones.

3. Database -- maintains information on machines, jobs, expected time for completion (ETC) data for all machine-job pair listings. These ETC's are used by the Scheduler when assigning machines to current jobs.
4. Learning and Accounting -- tracks and reports processes that exceed their ETC by more than a predetermined amount of time and may cause the schedule of other processes to be in danger of not being met. The Learning and Accounting process also updates the Database with the actual run-time of the process when run on the current machine [Ref. 2].

In order to come up with a good schedule, the SmartNet Scheduler uses data obtained *a priori* about the applications to be scheduled and the resources available. Thus, every new application-machine pair must be included in the SmartNet Database.

D. SUMMARY

As this chapter shows, the task of predicting application execution times is currently a research area that is very open. Most RMS's today must have the user of an application provide input data when the application is first executed in its computing environment. The predictions become more accurate only after the application has been executed several times on each machine available, making it difficult when the computing environment is very large. The research conducted recently at Ohio State University is step in the right direction for heterogeneous computing environments. This thesis, while focusing on a small section of the overall problem, is also a step in the right direction.

THIS PAGE INTENTIONALLY LEFT BLANK

III. APPROACH

In order to refine the existing task execution-time prediction model, it was necessary to break the model down into its component parts and analyze, correct, and validate each component. This chapter describes how the model is partitioned, and what steps were needed to modify the model from the original, incorrect model into the existing one.

A. DETERMINING THE ACCURACY OF THE TOOLS AND METHODS USED IN THE PREDICTION MODEL

As stated in Chapter II, the prediction model divides an application's execution time into its computation time and its communication time. From the experiments run during the previous work mentioned in Chapter II, we suspected that the model could accurately predict an application's computation time, but not its communication time. Since the research for this thesis meant to refine the previous model, it was necessary to validate that **both** the computation and the communication times were being modeled correctly.

1. Computation Time

The time that an application spends performing its CPU-intensive tasks is considered its computation time.

This parameter is known *a priori*, based on previous runs of the application, and represents the time the application is *expected* to use the CPU. This computation time, together with the time the application spends communicating, make up the application's total run time.

As was mentioned in Chapter II, the previous model ran an application that did not involve any communication, and used that execution time as the computation time for that application. If the same application is executed again, but this time needs to run through the same calculations two or more times, then the execution time observed for the first run would only need to be multiplied by two or more, depending on how many times the application expected to perform the calculations.

For example, the application emulator that was used to validate the prediction model used a computation thread that performed a matrix multiplication problem. If the application being modeled was to run through the matrix multiplication problem only once, then it would use the computation time that was already recorded from running the application previously. But if the application being modeled was expected to run through the matrix multiplication three times, then the computation time previously known would simply be multiplied by three. The question raised by this thesis was, "Is this the right way to obtain the computation time of an application?"

Intuitively, this method seems to be correct. The question then became whether the execution times obtained from the application emulator accurately represented the amount of time the application spent performing its matrix multiplication calculations. To answer this question, we needed to modify the application emulator slightly. Previously, the emulator measured how long the application took to execute. In order to run the emulator, the user started a "master controller" process that triggered a timer, started the application process, then when the application process reported to have completed, the master process would stop the timer. The problem with this method is that it did not take into account the amount of time spent assigning the application to a machine, nor the time it would take the application process to send its completion information back to the master process.

The application emulator was modified so that it would itself time how long it took in performing its CPU-intensive task. This value is then passed back to the master controller process to be output for analysis purposes. The master process now has data on both how long it took for the application to execute, and the duration of the CPU-intensive portion of the application.

2. Communication Time

The original prediction model divided an application's communication time into three parts:

a) The sender's time spent preparing its messages before sending them;

b) The time transmitting the messages;

c) The time the receiver spends processing the messages received.

Parts a) and c) are grouped together to make up the **latency time**. By measuring the **throughput** of a network link and knowing the size and number of messages to be sent, we can determine part b). If there is more than one process sharing a link between or within a machine, then the previous model simply divided the overall throughput of that link by the number of processes sharing it.

a) Throughput

As with the model's method of predicting an application's computation time, we also raised the question of whether its method of predicting the application's communication time was correct. The first step was to determine whether the model was using the correct values for the throughput between the three machines, and within each machine, in the test bed. This involved the following steps:

1) Review the code that was used to measure the throughputs for accuracy and possible mistakes;
2) Since the program measured throughput by sending a message of known size and timing how long it took, then see what the throughputs would be when sending a small message versus that of a large message;
3) Test the difference in throughput when only one process is transmitting versus when more than one process are sharing a link;
4) Validate the throughput-measuring program by using a commercially-available network analyzer to measure the throughputs.

b) Latency

The next step in determining the accuracy of the model when computing an application's communication time was to determine if the method used to compute its latency time was correct. This involved simply a review of the code used to measure the latency time. The method used for the measurement is as described in Chapter II. The values measured were so insignificant to the overall execution time that the latency time was left out of the model.

What the model failed to include was the latency time incurred when assigning a process to be executed on a remote machine (as mentioned in the previous section regarding "Computation Time"). It is possible that this was left out of this model because of the proximity of the three machines in the test bed, thus making this latency time very small. However, in a real distributed computing environment, the distance between processors can be very long. The latency involved in assigning a process to a distant machine would be a significant factor in the scheduling process.

B. ANALYZING THE MODEL AS A WHOLE

When devising a model to use, one chooses a measure to estimate. In this case, the measure was the time at which the last task of an application, when scheduled among one or more machines in a network, completes. In this thesis, as described in Chapter II, the test-bed of three homogeneous machines and one emulated application composed of three homogeneous processes allowed for 27 schedules to be modeled (see Table 4). Table 6 is an example of the input to the prediction model and the subsequent predicted execution time of Schedule One, in which all three processes are assigned to Processor One.

Each process is broken down into two parts, each representing the communications between that process and the other two processes in the simulated application. As mentioned earlier, the specific inputs to the model that needed to be reviewed for accuracy and validated were the following:

1) CPU time

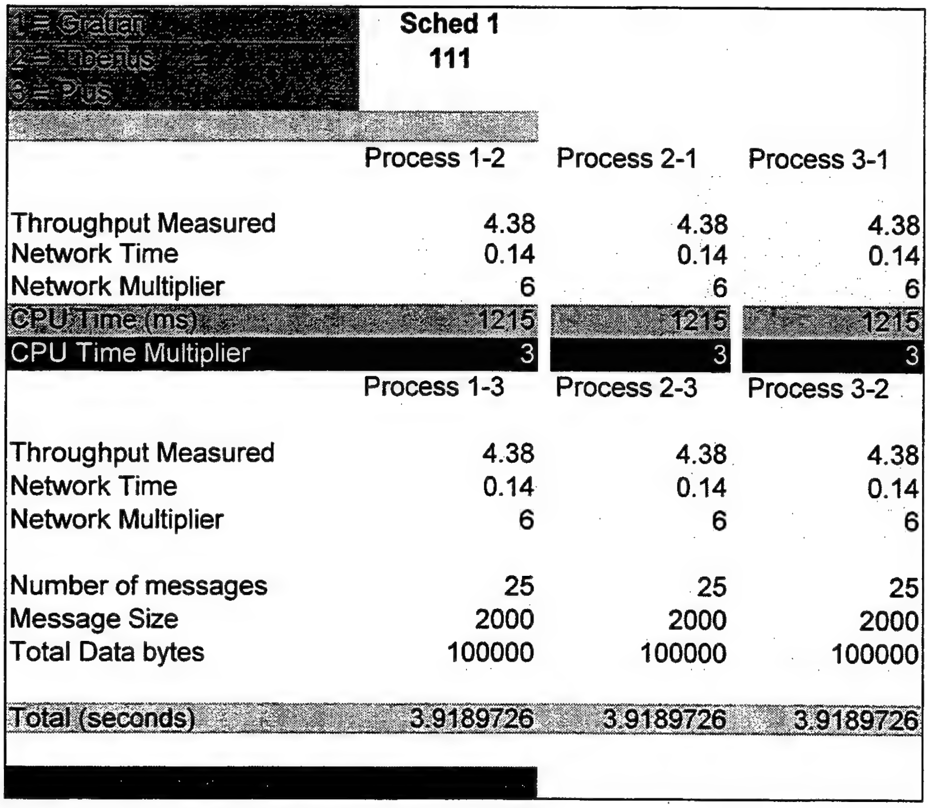
2) CPU time multiplier

3) Throughput

4) Network multiplier

5) Network Time.

The reasoning behind the use of the "network multiplier" input to the model is shown in Figure 7. The figure shows the three processes within the application, all scheduled to machine one (as is done in Schedule One). Since all three processes share the internal memory, or link, within the machine, the network multiplier is "six."



1 = Gratian 2 = Tiberius 3 = Pius	Sched 1 111		
	Process 1-2	Process 2-1	Process 3-1
Throughput Measured	4.38	4.38	4.38
Network Time	0.14	0.14	0.14
Network Multiplier	6	6	6
CPU Time (ms)	1215	1215	1215
CPU Time Multiplier	3	3	3
	Process 1-3	Process 2-3	Process 3-2
Throughput Measured	4.38	4.38	4.38
Network Time	0.14	0.14	0.14
Network Multiplier	6	6	6
Number of messages	25	25	25
Message Size	2000	2000	2000
Total Data bytes	100000	100000	100000
Total (seconds)	3.9189726	3.9189726	3.9189726

Table 6. Original Model's Prediction for Schedule 1

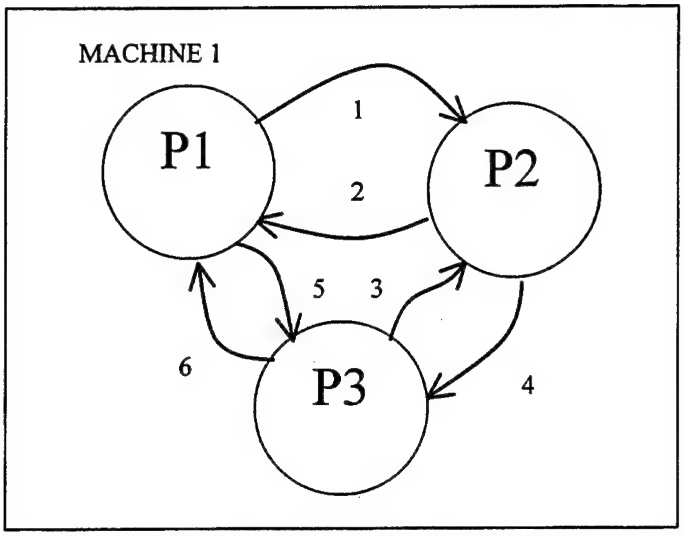


Figure 7. Sharing of a communication link within a machine.

If the schedule being modeled were Schedule Two, the first process would be scheduled on Machine Two, and the other two processes would be scheduled on Machine One. As shown in Figure 8, the Ethernet link between machines one and two would be shared by four communicating threads, and the internal link of machine one would be shared by two communicating threads.

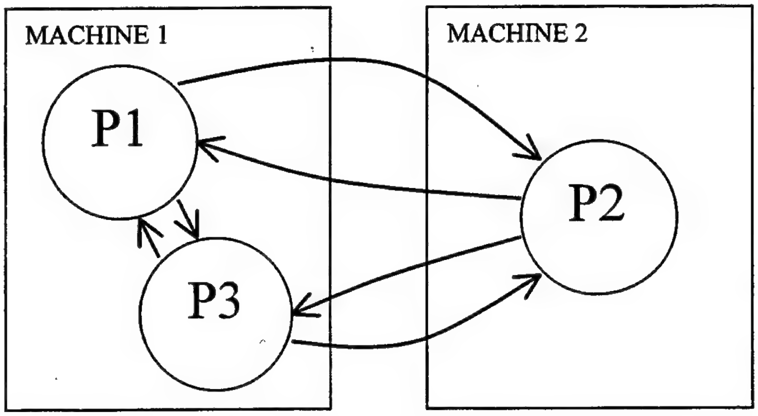


Figure 8. Sharing Communication Links Between Two Machines.

So far, we have shown examples of an instance when all processes are scheduled to one machine, and an instance when one process is scheduled to one machine and the other two processes are scheduled to another machine. The third example is that of each process being scheduled on a separate machine. Figure 9 shows how Schedule Six shares its communication links. In this instance, the model used a network multiplier of six to represent the three machines sharing the Ethernet link.

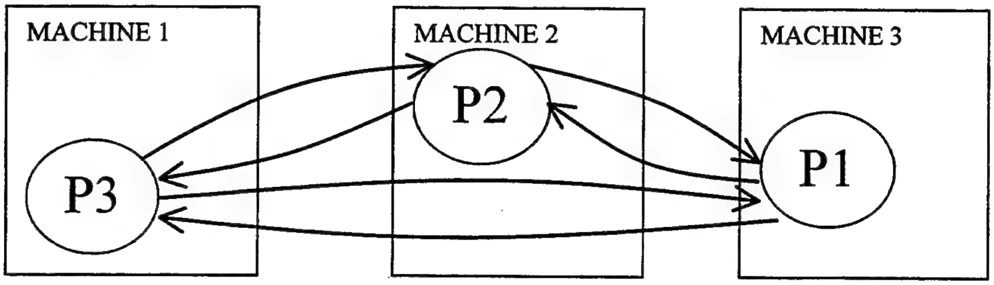


Figure 9. Sharing Communication Links Between Three Machines

C. VALIDATION OF NEW MODEL

After making the necessary changes to the prediction model, it needed to be validated using the application emulator, just as the original prediction model was validated with the emulator. As outlined in Chapter II, the emulator takes the model parameters as input and runs an application using those parameters. For example, if the input parameters to the model were such that the CPU multiplier were six, the number of messages to be passed between processes was 2000, and the size of the messages was 4000 bytes, then the application emulator would run through the matrix multiplication problem six times, send 2000 messages between each process, and each message would be 4000 bytes long. The actual run time of the application is then be compared to the output of the model.

D. SUMMARY

The coarse-grained, simple approach of the original prediction model proved that its methods were not "good enough" to accurately predict the execution times of a particular type of application. This chapter outlined the steps taken to determine which methods within the model needed modification. The model is broken down into an application's computation time and its communication time. The combination of the two times makes up an application's execution time.

The following chapter contains details about the experiments that were conducted in order to answer all of the questions posed in Chapter III. The results of the experiments and their meaning will show our reasoning behind the changes that were made to the original prediction model.

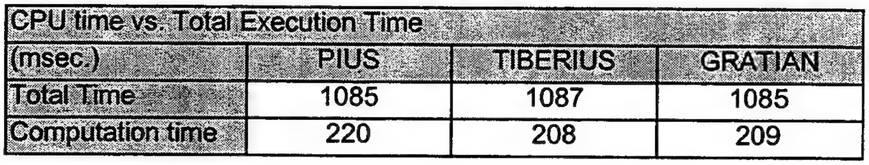
IV. REFINING THE PREDICTION MODEL

Refining the execution-time prediction model required partitioning the model into its components and validating or modifying each one. This chapter explains in detail how the model was partitioned and what experiments were conducted to ensure each component was correct. Our approach to this problem included dividing the model into its computation time inputs and its communication time inputs.

A. MEASURING CPU TIME

In order to measure the exact amount of time an application spends performing its CPU-intensive tasks, it is necessary to run that application so that it does not perform any communication. In the case of the application emulator used in this thesis, we placed a timer around the code that performed the matrix multiplication problem that made up the CPU-intensive part of the application. In this way, it was possible to single out that part of the application, and to know exactly how long the CPU spent executing it.

After inserting the extra lines of code into the application emulator, we ran the emulator with several different input parameters. The goal was to run the application emulator on each of the three machines separately, so that each machine executed the matrix multiplication problem without ever needing to communicate with other processes. After running the emulator on all three machines, we were able to compare the computation time versus the total execution time of the application on each of the three machines. The results are shown in Table 7.



CPU time vs. Total Execution Time			
(msec.)	PIUS	TIBERIUS	GRATIAN
Total Time	1085	1087	1085
Computation time	220	208	209

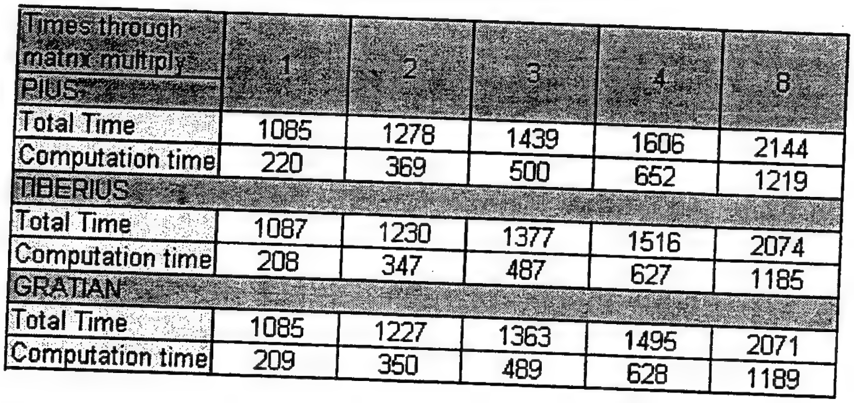
Table 7. CPU Time vs. Total Execution Time

As Table 7 shows, the time spent in the actual matrix multiplication was approximately one-fifth of the total execution time of the application. Since the application did nothing other than compute the matrix multiplication, then the rest of the execution time is the time spent transmitting the schedule from the master-controller process to each of the three machines.

B. DETERMINING THE CPU MULTIPLIER

Since we now know how long the application emulator spends computing the matrix multiplication problem, we needed to execute the application such that it ran through the problem more that one time. The input parameters to the application emulator include an input to how many times the application is to run through its CPU-intensive portion, and therefore, it needed to be modeled.

Once again, the method used was to run the application emulator on one machine at a time, and having it send zero messages. The emulator was executed several times, each time incrementing the input that determined how many times the application would perform the matrix multiplication problem. Table 8 shows the outcome of the experiments.



Times through matrix multiply	1	2	3	4	8
PIUS					
Total Time	1085	1278	1439	1606	2144
Computation time	220	369	500	652	1219
TIBERIUS					
Total Time	1087	1230	1377	1516	2074
Computation time	208	347	487	627	1185
GRATIAN					
Total Time	1085	1227	1363	1495	2071
Computation time	209	350	489	628	1189

Table 8. CPU Multiplier Experiment Results

Figure 10 shows that the computation times do in fact grow linearly, but not symmetrically. For example, if the application emulator took 220 msec. to execute the matrix multiplication once, then the previous model assumed that it would take 440 msec. to execute it twice, 660 msec. to execute it three times, etc. Our experiment proved this assumption to be incorrect. The formula that the previous model used to compute the CPU time was as follows:

$$t \approx (CPUtime) * (TimesThroughMatrix).$$

The correct way to compute the CPU time, when the application is to run through the matrix computations more than once, is to use the following formula:

$$t \approx (CPUtime) * (TimesThroughMatrix \div 1.3).$$

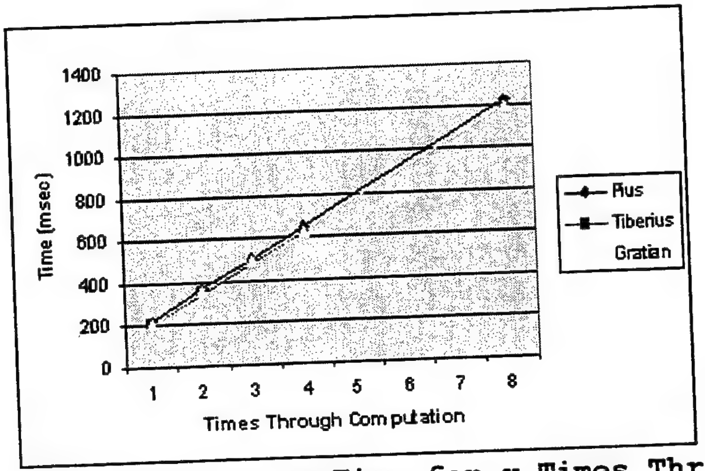


Figure 10. Computation Time for x Times Through Multiplication Matrix

C. MEASURING THROUGHPUT

1. Analyzing the Code Used for Measurement

Measuring the correct throughput of the network and the throughput within a machine is a crucial step in the prediction model. The very first step in determining whether the model was using the correct throughput was to review the code used for the measurement. A thorough review showed that the code used to measure the throughput for the original prediction model was off by a factor of two.

The basis behind the program used to measure throughput was to measure the amount of time spent sending a message of a given length to a given IP address and having the receiving machine echo the message back to the sending machine. For example, to measure the throughput between machine 1 and machine 2, machine 1 sends a message of 2000 bytes to machine 2, which in turn echoes the message back to machine 1. If machine 1 times how long this sending and receiving took and divides that roundtrip-time by twice the size of the message, it should come up with a value that represents the throughput between the two machines. In order to get statistically correct data, the test is performed 5000 times. The following pseudo-code outlines the method used:

```
begin
     numberOfBytes = sendString.length() * 2;
     for (1 through 50){
          dataTime = 0;
          totalDataBytes = 0;
          start_Time;
               for (1 through 100){
                    send (sendString);
                    receive (echoedString);
               }
          stop_Time;
          dataTime = stopTime-startTime;
          totalDataBytes = numberOfBytes * 100 * 2;
          throughput = totalDataBytes / dataTime;
     }
     throughput = AVERAGE(throughputs 1 through 50);
end
```

If the number of bytes being sent is calculated as the length of the string multiplied by two (due to the echoing of the string from receiver to sender), then it is not necessary to multiply "totalDataBytes" by two also. The effect that this error in the code had was to give the impression that the throughput was twice as much as was actually measured. The error was simply fixed by removing the final multiplication by two from "totalDataBytes."

2. Measuring Throughput with Messages of Different Sizes

The original prediction model used one message size to measure the throughput of a network. It measured the throughput using the program described in the previous section, once for a machine's internal link, and again for an Ethernet link between machines. The model assumed that the throughputs would be the same no matter what the size of the message. This thesis proved that assumption to be incorrect. Figure 11 shows the results of an experiment run which measured the throughputs within and between the machines in the test bed. The experiment used several different-sized messages to test if the throughput was affected by message size. Each time the experiment was run, it would only run between one set of machines at one. For example, when testing the throughput between machine 1 and machine 2, the experiment did not send any other messages besides those required to test that throughput. This ensured that nothing was affecting the measure of throughput besides the messages intended for that purpose.

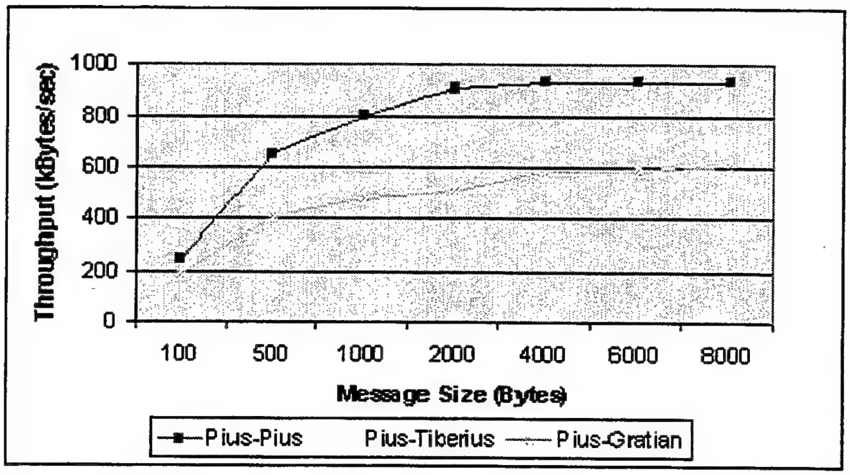


Figure 11. Throughput vs. Message Size for Various Links

D. DETERMINING THE NETWORK MULTIPLIER

An application may send messages either between processes within a single machine, or between processes residing on different machines. In order for our prediction model to accurately predict the amount of time it takes for these messages to be transmitted between processes, it must know how the network links between the machines handle several processes communicating concurrently.

To perform interprocess communication, there are several ways that the processes can send and receive the messages. In order to choose the right method of communication, one has to take into account the type of environment under which the processes will be communicating. Some methods make use of shared memory space between the processes while other methods may not be supported by all of the systems making up the distributed system. [Ref. 4]

The method explored by this thesis is the use of Java™[1] sockets. There are two forms of sockets that can be implemented, depending on the application using them. The first is a stream socket, which is implemented in the TCP/IP protocol. These sockets allow for reliable, connection-oriented communications. The second type of socket is a datagram socket, which is implemented in UDP/IP. This is a connectionless form of communication, and thus is not always reliable, but it is more efficient. Because of the stream socket's attributes, it seemed to be the best choice for our application emulator. [Ref. 4]

Several experiments were run in order to determine how the throughputs that were measured previously (outlined in the last section) were affected when more than one process was transmitting on a link at once. The experiments are presented in three sections, those run within a machine, those run between machines, and those run with a combination of the two. In order to attempt to keep things simple, all of the experiments were limited to sending messages of sizes 2000 or 4000 bytes.

1. Within a Machine

The throughput between processes communicating within a machine was tested by measuring that throughput when only one process is sending messages to another process. After observing what this throughput was, other experiments were conducted where two, three, and four processes were sending messages within the same machine. By comparing the differences in throughput, we were able to see what kind of network multiplier we should be using in the prediction model.

The original model assumed that if two processes were sharing a communications link, then the network multiplier would be two. If three processes were sharing the link, the multiplier would be three, and so on. These experiments prove that assumption to be false.

The layout of the experiments are as shown in Figure 12 and Figure 13. The results are shown in Table 9.

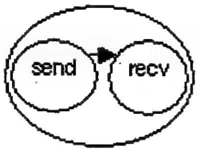


Figure 12. One Process Sends Messages to Another Process Within the Same Machine

[1] Java™ is a Trademark of Sun Microsystems.

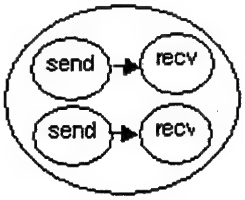


Figure 13. Two Processes Transmitting Within the Same Machine

	Number of Processes Transmitting			
Msg Size	1	2	3	4
2000 Bytes	1	1.2	1.5	1.7
4000 Bytes	1	1.2	1.6	1.7

Table 9. Network Multiplier Within a Machine

2. Between Machines

The same type of experiments that were conducted in order to determine the network multipliers within a machine were conducted between machines. The same assumptions and conditions apply in the case of IPC between machines as within a machine (because of the use of sockets). Figure 14 shows the experiments that were conducted. The results of the experiments are shown in Table 10 and Table 11.

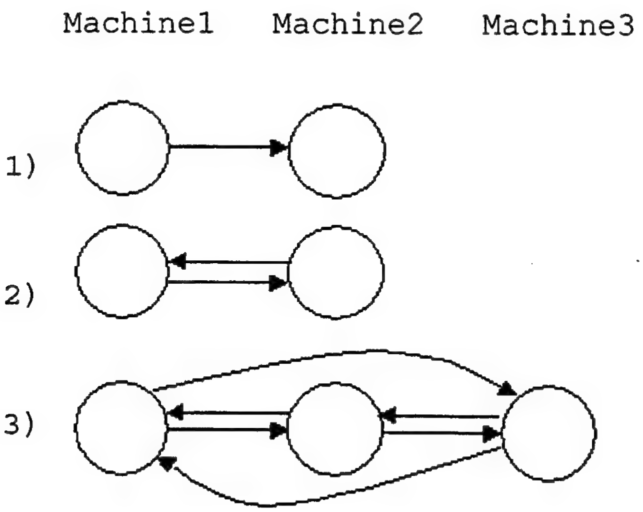
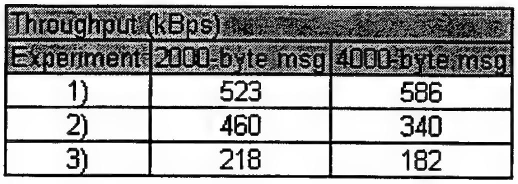
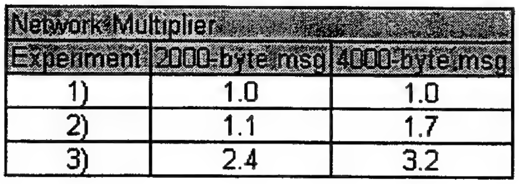


Figure 14. Experiments 1-3 Conducted Where Processes Transmit Between Machines



Throughput (kBps)		
Experiment	2000-byte msg	4000-byte msg
1)	523	586
2)	460	340
3)	218	182

Table 10. Throughput Measured for Experiments 1-3



Network Multiplier		
Experiment	2000-byte msg	4000-byte msg
1)	1.0	1.0
2)	1.1	1.7
3)	2.4	3.2

Table 11. Network Multipliers from Experiments 1-3

3. A Combination of Transmitting Between Processes Between Machines and Within a Machine

The final scenario posed by this thesis concerning throughput in a network is that of several processes communicating, some within the same machine and others on another machine. This experiment was conducted to see the difference between this scenario and that of keeping all processes either on separate machines or in the same machine. The experiment conducted is shown in Figure 15. Table 12 shows the results.

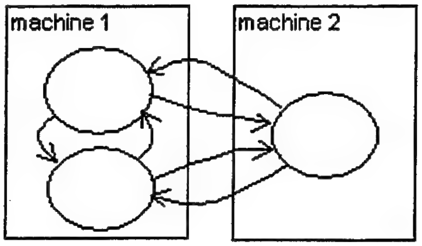
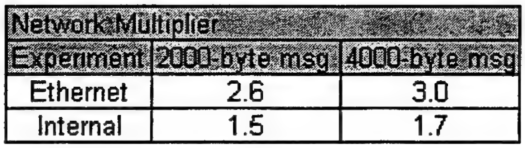


Figure 15. Experiment With Three Processes on Two Machines



Network Multiplier		
Experiment	2000-byte msg	4000-byte msg
Ethernet	2.6	3.0
Internal	1.5	1.7

Table 12. Network Multipliers for Internal and External Links

The previous model assumed a simple formula, once again, of setting the network multiplier to the different number of transmissions on a single link. In this example, it assumed that the internal link on machine 1 would have a network multiplier of two. The external link, or Ethernet connection, would have a multiplier of four. This experiment proved that assumption to be incorrect.

E. USING A NETWORK ANALYZING TOOL

While running the experiments to measure the throughput of the network, it was necessary to verify that these measurements were correct. In order to do this, a commercially-available LAN analyzer was connected to the network and set to measure the traffic being transmitted. The tool that was chosen was Network Instruments' Observer™. Observer™ assists the user in isolating parts of a network and view exactly what is being transmitted, which protocols are being used, the rate at which the data propagates, how many errors occur, and many other performance-related measures.

To validate the output of the throughput-measuring application written for this thesis, Observer™ was set up to monitor the transmissions over the network. Its measurements were compared with the output of our application while our experiments were being conducted. Observer™ consistently matched our application's output, therefore validating that the results of our experiments were in fact correct.

F. SUMMARY

In order for the execution-time prediction model to accurately predict the run-times of an application when it is distributed across a network, the model must have correct input as to the throughput of the network, the throughput within each machine to be scheduled, and the amount of time the application expects to spend performing CPU-intensive tasks.

The previous model made many assumptions that this thesis has proven to be incorrect. Chapter V will present in detail how the new methods used in the prediction model were validated by using the application emulator and comparing actual versus predicted run-times.

THIS PAGE INTENTIONALLY LEFT BLANK

V. RESULTS

This chapter presents the results of several experiments that were conducted in order to validate our new prediction model. The results of the experiments, which were obtained by running the application emulator with various input parameters, are compared with the output of the model.

The test bed of three Pentium machines that was used to validate the original model was not modified for this thesis. All of the experiments conducted were on the Microsoft Windows NT Workstation 4.0 operating system, and all of the applications were run in an asynchronous, non-GUI mode. For each experiment, we varied one or more of the following input parameters: the number of messages to be transmitted between processes, the size of those messages, the amount of computation to be done by the processes, or a combination of any of these.

Our model will only be successful if it chooses the best schedule among the 27 possible schedules of the three processes among the three machines. If the model predicts the execution times correctly, then it will in fact choose the correct schedule. It should also predict when one schedule is relatively better or worse than another schedule.

A. EXPERIMENT ONE: 25 MESSAGES, 2000 BYTES, 1 MATRIX MULTIPLICATION

Our model does not predict absolute run-times correctly, but it does predict relative run-times correctly. Schedules 6, 8, 12, 16, 20, and 22 are the fastest schedules, and schedules 1, 14, and 27 are the slowest. Our model does not show as much difference between the schedules' predicted run-times as the actual run-times seem to indicate, but it does correctly distinguish between the slowest and the fastest schedules. Figure 16 shows the results of this experiment.

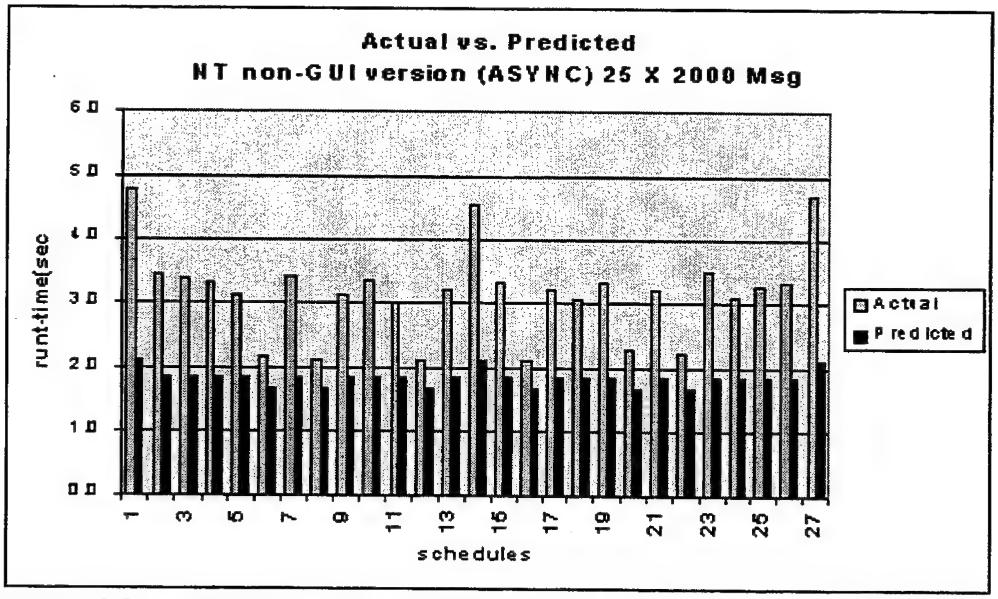


Figure 16. Actual vs. Predicted Run-Times for Experiment One

B. EXPERIMENT TWO: 25 MESSAGES, 20K BYTES, 1 MATRIX MULTIPLICATION

Our model does not predict absolute run-times correctly, nor does it predict relative run-times correctly. The model does a good job of accurately predicting the run-times for schedules 6, 8, 12, 16, 20, and 22, but it also shows those schedules as not being the fastest when in fact they are. Our model, once again, does not show as much difference between the schedules' predicted run-times as the actual run-times seem to indicate. Figure 17 shows the results of this experiment.

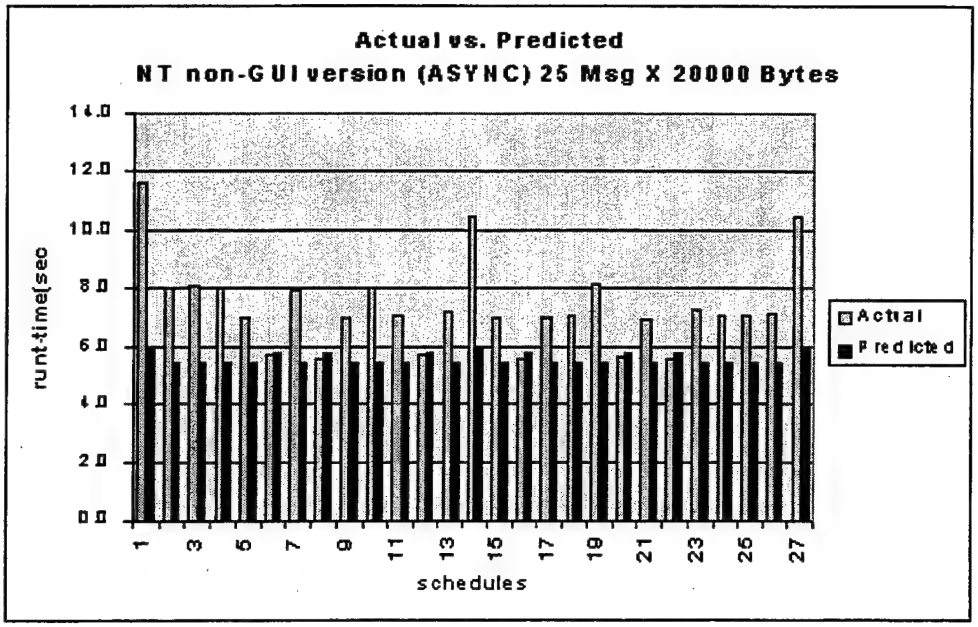


Figure 17. Actual vs. Predicted Run-Times for Experiment Two

C. EXPERIMENT THREE: 250 MESSAGES, 200 BYTES, 1 MATRIX MULTIPLICATION

In contrast to the last two experiments which dealt with a small number of large-sized messages, this experiment dealt with a medium number of small-sized messages. The model once again did a good job of predicting the slowest and the fastest schedules. It predicted the relative run-times well, and it was closer than the last two experiments in predicting the absolute run-times, but it still lacks a high level of accuracy. The results are shown in Figure 18.

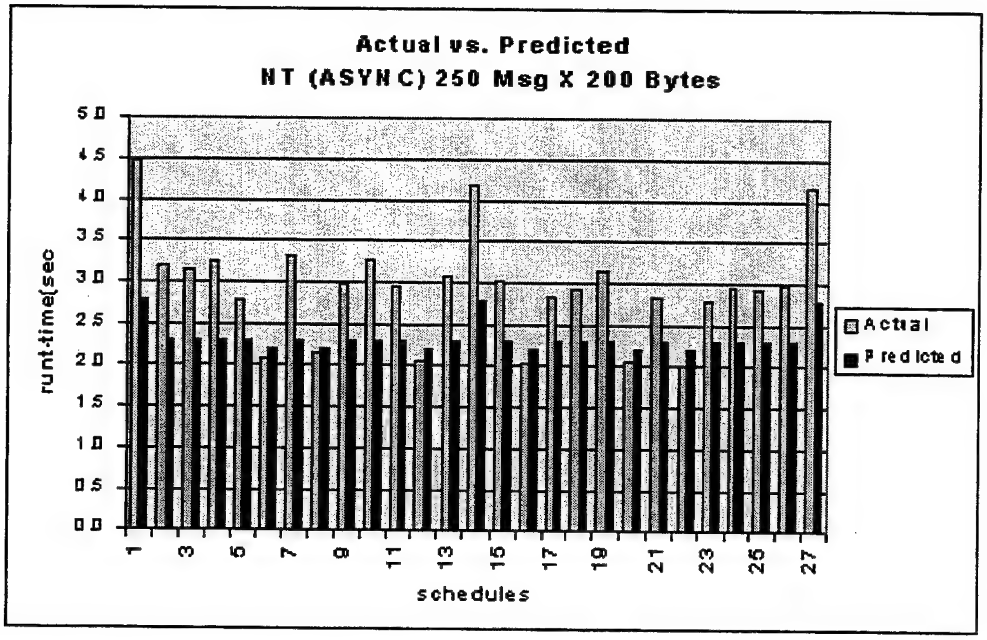


Figure 18. Actual vs. Predicted Run-Times for Experiment Three

D. EXPERIMENT FOUR: 1250 MESSAGES, 200 BYTES, 1 MATRIX MULTIPLICATION

For this experiment, we increased the number of messages over the number used in experiment three, and kept the size of the messages the same. Here we see that the model did not accurately predict either the relative, nor the absolute run-times of the schedules. The model is very close in predicting the run-times of all schedules except those of 6, 8, 12, 16, 20, and 22, where all three processes are on different machines. Figure 19 shows the results.

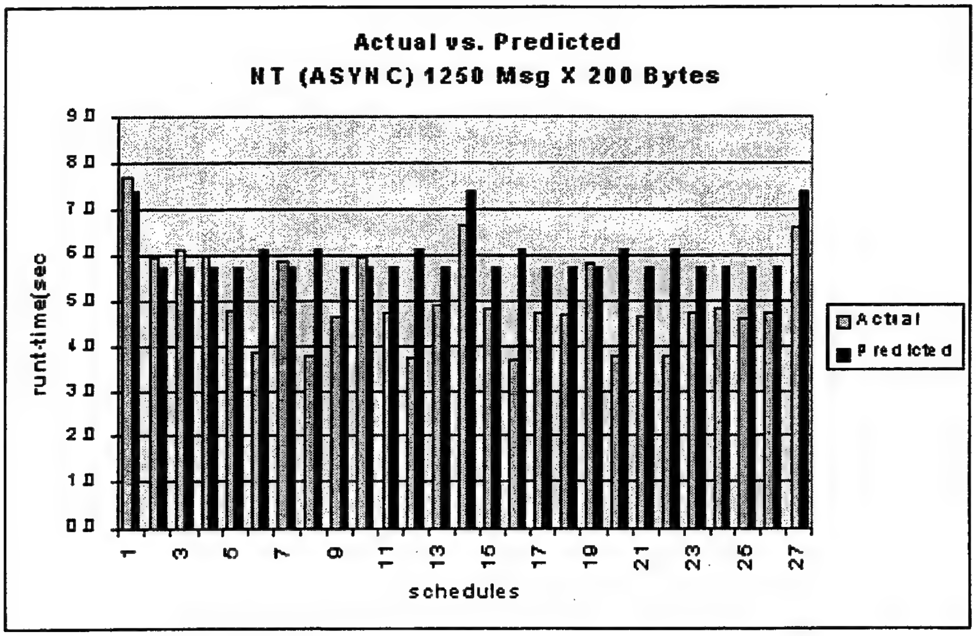


Figure 19. Actual vs. Predicted Run-Times for Experiment Four

E. EXPERIMENT FIVE: 100 MESSAGES, 2000 BYTES, 1 MATRIX MULTIPLICATION

In this experiment, the model did not predict the run-times very well. It did not show a significant difference between any schedules except 1, 14, and 27. In those three schedules, where all three processes were assigned to the same machine, the model correctly shows the run-times as being greater than all of the other schedules. The results are shown in Figure 20.

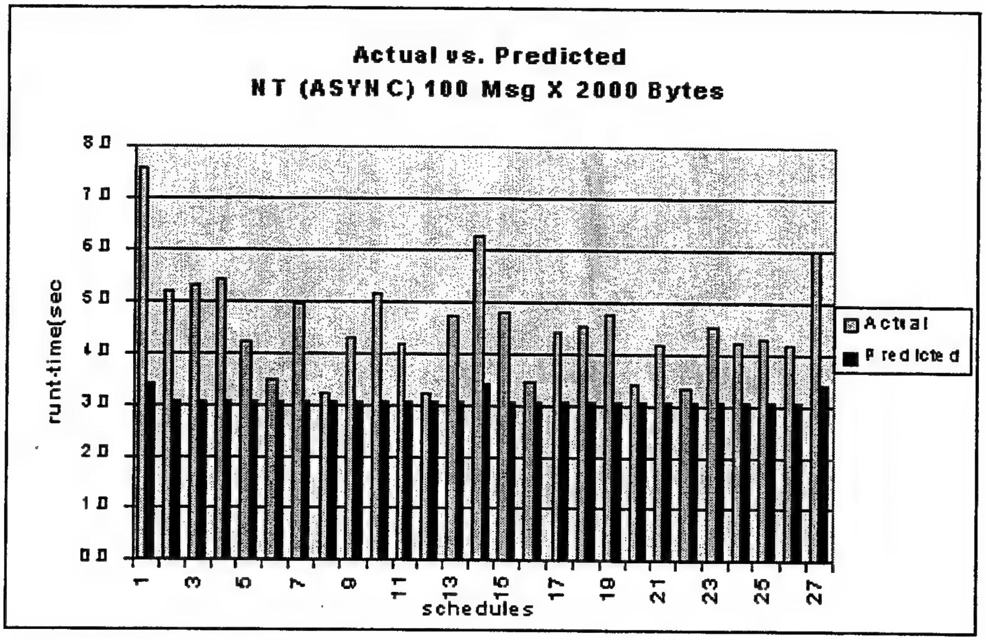


Figure 20. Actual vs. Predicted Run-Times for Experiment Five

F. EXPERIMENT SIX: 250 MESSAGES, 2000 BYTES, 1 MATRIX MULTIPLICATION

The model accurately predicts the run-times of various schedules in this experiment. The model predicted very accurately the run-times of the schedules in which all three processes were assigned different machines (schedules 6, 8, 12, 16, 20, and 22). It also was accurate in predicting when one process was on one machine and the other two processes were on another machine. The only discrepancy, which is actually a factor in all of our experiments, was that machine one was also assigned to run the master controller process, so it took up CPU time that was not included in our model. This discrepancy is what caused the "spikes" in schedules number 1, 2, 3, 4, 7, 10, and 19. The model was not as accurate in predicting the absolute run-times of schedules 1, 14, and 27, when all three processes are assigned to one machine, but it did correctly show these schedules as being slower than the others. The results are shown in Figure 21.

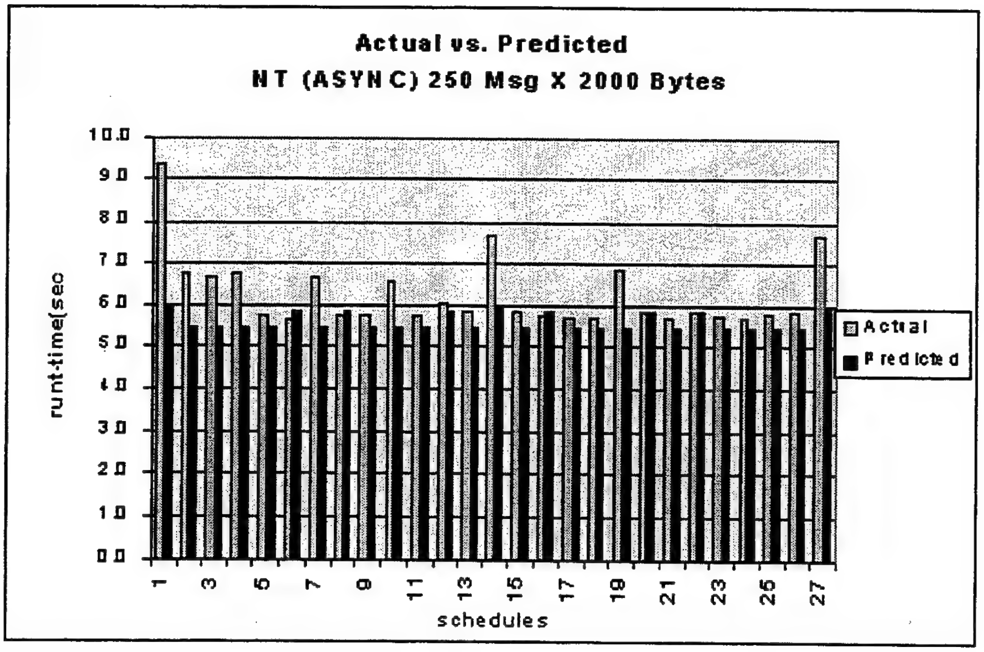


Figure 21. Actual vs. Predicted Run-Times for Experiment Six

G. EXPERIMENT SEVEN: 1250 MESSAGES, 2000 BYTES, 1 MATRIX MULTIPLICATION

In this experiment, for which the number of messages and the size of those messages were large, the model did very well. The absolute run-times of the model versus the application were very close, and all of the schedules' run-times were relatively correct. The results are shown in Figure 22.

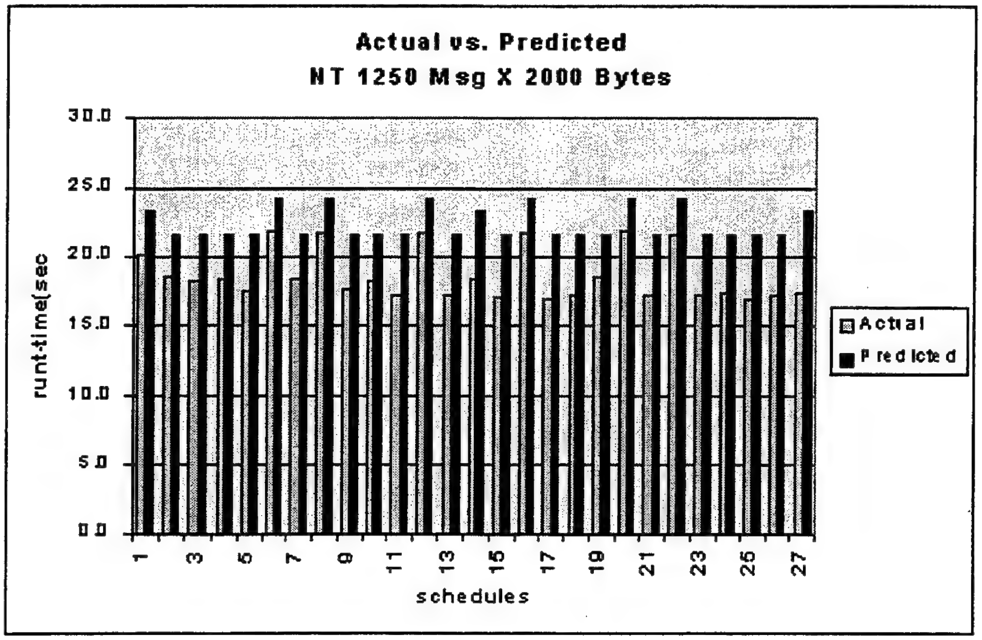


Figure 22. Actual vs. Predicted Run-Times for Experiment Seven

H. EXPERIMENT EIGHT: 1250 MESSAGES, 4000 BYTES, 1 MATRIX MULTIPLICATION

Once again, like the previous experiment, the model did well in predicting the run-times of the schedules. It was not completely accurate in predicting the absolute performance, but it came very close. It predicted the relative performance of the schedules very well. The results are shown in Figure 23.

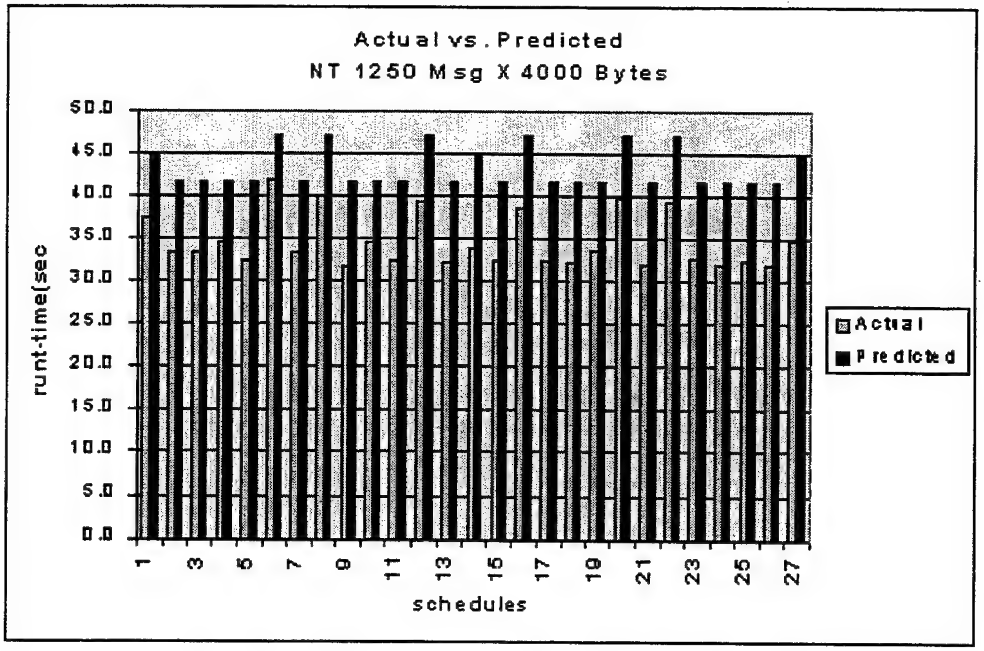


Figure 23. Actual vs. Predicted Run-Times for Experiment Six

I. EXPERIMENT NINE: 2500 MESSAGES, 5000 BYTES, 6 MATRIX MULTIPLICATIONS

As the number of messages and the message size continue to grow, the model's error margin also continues to grow. The output of the model is no longer as accurate as in experiment eight, but it still does a very good job of predicting the relative run-times of the schedules. Figure 24 shows the results of this experiment.

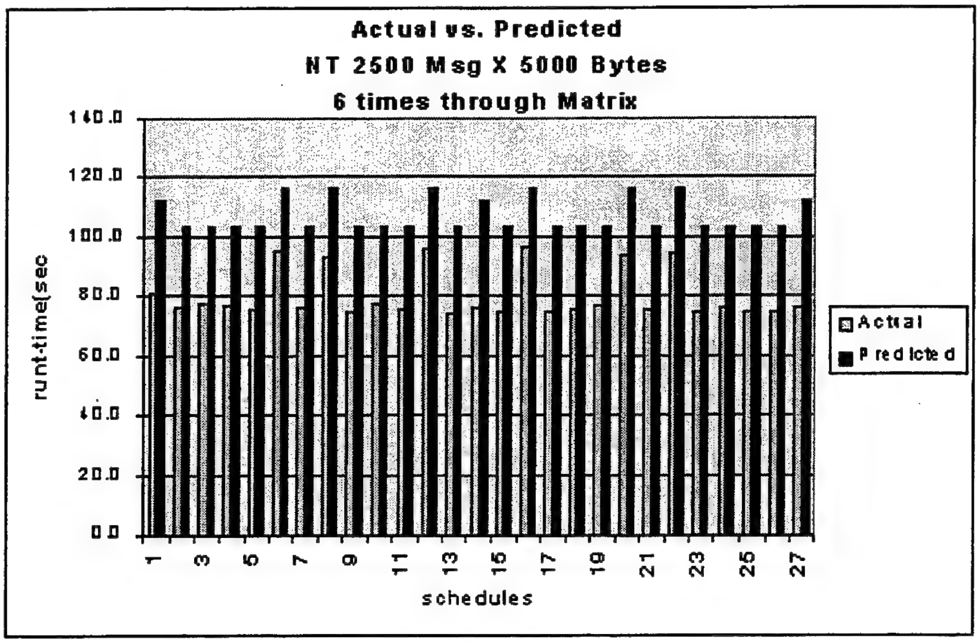


Figure 24. Actual vs. Predicted Run-Times for Experiment Eight

J. SUMMARY

The results of nine experiments conducted to validate our model were presented in this chapter. The goal of the experiments was to vary the input parameters to the model and to the application emulator so that we would get a mix of compute-intensive, communication-intensive, and both compute- and communication-intensive applications in our experiments.

The results show several things:

1. Most of the inputs to the model were gathered from running experiments sending many messages of size 2000 or 4000 bytes. This fact made itself evident

that experiments six, seven, and eight provided the most accurate predictions of relative and actual performance.

2. Where the original model did a good job of predicting absolute run-times of applications that were computation-intensive, this thesis did not do such a good job. However, since this thesis proved that the methods that the previous model was using were incorrect, there is still something that is not being accounted for in the execution-times of those applications.
3. The model is valid (predicts relative performance well) for communication-intensive applications.

Chapter VI gives a complete summary of this thesis, including some ideas on future work to be conducted in this area of research.

VI. SUMMARY

Nowadays, it is common to see the use of a network of machines to distribute the workload and to share information between machines. In these distributed systems, the scheduling of resources to applications may be accomplished by a Resource Management System (RMS).

In order to come up with a good schedule for a set of applications to be distributed among a set of machines, the scheduler uses a model to predict the execution time of the applications. The model used for this thesis, which estimates the time that the last task will be completed when scheduling several tasks among several machines, uses an analytical, closed-form solution to solve the problem.

A previous thesis investigated questions similar to those in this thesis, and the model that it presented was a simple, coarse-grained model. While attempting to assess the detail of the model needed to come up with a good schedule, this previous thesis determined that its model was not detailed enough to provide an accurate prediction of the run-times of certain applications, mainly communication-intensive applications, to be scheduled by a RMS. The model needed to be refined in order to be usable.

A. FUTURE WORK

This thesis refined the original prediction model by using data collected within a controlled environment. This environment consisted of a test-bed of three Pentium machines configured exactly alike. Different distributed environments have different throughputs, latency, CPU speeds, varying operating systems between machines, and many other varying parameters. In order to test the model further, it would need to use input data from a variety of distributed environments.

The model also needs to be modified so that it can easily take input parameters from non-homogenous applications. Currently, the model predicts the performance of applications consisting of three homogeneous processes. This means that the processes all spend the same amount of time computing and they send the same number of equal-sized messages.

Another area that needs to be researched further is to test the refined model for accuracy when predicting synchronous applications. When our model was validated, it was only done so against applications that executed in an asynchronous fashion. The previous model was validated against both synchronous and asynchronous applications, and it found that it was completely inaccurate when predicting the run-times of synchronous applications.

B. CONCLUSIONS

As stated in Chapter IV, in order for the execution-time prediction model to accurately predict the run-times of an application when it is distributed across a network, the model must have correct input as to the throughput of the network, the throughput within each machine to be scheduled, the amount of time the application expects to spend performing CPU-intensive tasks, and the correct measure of dilation of CPU and network resources.

In order to provide the correct inputs for all of the parameters mentioned, we had to run extensive experiments within our test-bed of machines. Because of the need to run the sort of experiments that were outlined in Chapters III and IV, the model may be too detailed, and not accurate enough, to be efficient when providing input to a scheduler in an RMS.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1. Debra Hensgen, Taylor Kidd, David St. John, Matthew C. Schnaidt, H. J. Siegel, Tracy Braun, Jong-Kook Kim, Shoukat Ali, Cynthia Irvine, Tim Levin, Viktor Prasanna, Prashanth Bhat, Richard Freund, and Mike Gherrity, *An Overview of the Management System for Heterogeneous Networks (MSHN)*, 8th Workshop on Heterogeneous Computing Systems (HCW '99), San Juan, Puerto Rico, Apr. 1999,

2. P. Carff. When is a simple Model adequate for use in scheduling in MSHN? Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1998.

3. Michael A. Iverson, Fusun Ozguner, and Lee C. Potter. Statistical Prediction of Task Execution Times Through Analytic Benchmarking for Scheduling in a Heterogeneous Environment. *Proceedings of the IEEE Eighth Heterogeneous Computing Workshop (HCW '99)*, pages 99-110, April 1999.

4. A.M. Law and W.D. Kelton. *Simulation and Modeling Analysis*. McGraw-Hill, Inc., New York, second edition, 1991.

5. Doreen Galli. *Distributed Operating Systems*. Prentice Hall, Inc., New Jersey, 1998.

6. Ingrid Bucher, Rebecca Koskela, Margaret Simmons. *Instrumentation for Future Parallel Computing Systems*. ACM Press, New York, 1989.

7. Andrew Tanenbaum. *Computer Networks*. Prentice-Hall, Inc., New Jersey, third edition, 1996.

8. Javasoft. Java development kit documentation. Manual, 1998. Available at http://www.java.sun.com/products/jdk/1.1/download-pdf-ps.html.

9. Scott Oaks, Henry Wong. *Java Threads*. O'Reilly and Associates, Inc., California, 1997.

10. Sidnie Feit. *TCP/IP: Architecture, Protocols, and Implementation with Ipv6 and IP Security*. McGraw-Hill, Inc., New York, 1999.

11. Merlin Hughes, Michael Shoffner, Derek Hamner. *Java Network Programming*. Manning Publications Co., Connecticut, 1999.

INITIAL DISTRIBUTION LIST

No. copies

1. Defense Technical Information Center 2
8725 John J. Kingman Road, Ste 0944
Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
Naval Postgraduate School
411 Dyer Rd.
Monterey, CA 93943-5101

3. Chairman, Code CS 1
Naval Postgraduate School
Monterey, CA 93943-5101

4. Prof. James Bret Michael 1
Computer Science Department Code CS
Naval Postgraduate School
Monterey, CA 93943-5000

5. Prof. Mantak Shing 1
Computer Science Department Code CS
Naval Postgraduate School
Monterey, CA 93943-5000

6. Blanca A. Shaeffer 1
12 E Quincy #2
Riverside, IL 60546